Rapira Reborn

A Guide to the Soviet-era Programming Language from the 80s

Amanvir Parhar

Table of Contents

1. An Introduction to Rapira

2. Getting Started

3. Variables

4. Tuples

5. Strings

6. `If/else` statements

7. Loops

8. Functions

9. Challenges

10. Sample Solutions for Challenges

An Introduction to Rapira

History

Rapira (Russian for "rapier", a thin sharp sword) began as an educational programming language desgined by Andrey Ershov in 1982. It was desgined to serve as a bridge between Robic, the Soviet programming language used to teach the basics of computing at the primary school level, and more mature programming languages and ideas. Along with some other software, including text editors and interpreters, Robic and Rapira became part of the "Shkol'nitsa" ("Schoolgirl"), a software system for the Agat computer designed specifically for educational use. It was the Agat's relatively inexpensive nature that led the machine to become widely adopted in Soviet educational institutions.

The Language

Rapira is an interpreted language with dynamic typing.

Those who have experience with Pascal might notice syntactical similarities it shares with Rapira. This is because Rapira's design was strongly influenced by ALGOL, as well as other languages like POP-2 and SETL, which Pascal also draws inspiration from. From SETL, Rapira borrows its rather unique data type: the tuple. These are lists that contain elements of any type that can be of varying length, and the language provides a list of useful methods for indexing, comparing, or creating sublists of tuples.

Quite unsurprisingly, Russian was the only language Rapira supported at its conception, but support for English (as well as Romanian and Moldovan) has been added since then.

Why Rapira?

Firstly, Rapira proves to be an exemplary language to learn the basics of programming. Its simple and elegant syntax, which bears a striking resemblance to the Python programming language, and obviously, its origins as a tool to teach programming, make it effective for educational use even in the 21st century.

But even if you have been programming for a while, you'll still find this book useful. Why?

Well, I believe that it's imperative for us, the future generation of programmers, to preserve computing history. If no one is to learn the features of Rapira (or any other piece of software for that matter), we'd let vital parts of the journey that brought computers this far go unarchived and forgotten. So, the onus is collectively on us: we must take the time to educate ourselves on this old and beautiful Russian programming language, much like people would with endangered spoken languages in the real world.

That's why I set out to write this book. It'll serve as the world's first extensive, written documentation of the Rapira programming language.

I hope you'll enjoy this journey.

Getting Started

Setting up your development environment

First, since Rapira is an interpreted language, we need an interpreter to run some Rapira code!

Thankfully, Matthew Mikolay has created Rapture, an interpreter for Rapira that we'll be using throughout the course iof this book.

To get started with Rapture, you'll first need to have Git and Gradle installed. Let's get started with some Git installation instructions for macOS, Windows, and Linux (feel free to skip this section if you already have Git on your machine).

Installing Git

We need Git, a source code management tool, to clone the Rapture repository from GitHub.

macOS

There are several ways to install Git on a Mac. Running the git command itself on a Mac should run an installer within the Terminal itself.

```
$ git --version
```

Alternatively, you can use Homebrew to install git with:

```
$ brew install git
```

Windows

On Windows, the most straightforward way to download Git would be through the official build: https://git-scm.com/download/win (you have the option to choose between a 32-bit and 64-bit installer).

Alternatively, if you have Chocolatey, run this command to install Git Chocolatey package:

```
$ choco install git
```

Linux

There's a good chance that Git is already installed on your machine. If it's not, you can generally install Git through a package manager. If you���re on Fedora/Rhel/CentOS, you can use dnf:

```
$ sudo dnf install git-all
```

If you���re on a Debian-based distribution, such as Ubuntu, try apt:

```
$ sudo apt install git-all
```

If your distro wasn't covered here, I'd wager that, as a Linux user, you probably know what to do :wink:.

Installing Gradle

To build an executable to run Rapira code, we need to install Gradle. Gradle requires Java to be installed. Run the following command to confirm its installation:

```
$ java -version
```

macOS

You can run this one-line Homebrew command to install Gradle on macOS:

```
$ brew install gradle
```

Windows

Download the latest Gradle release from https://gradle.org/releases.

Unzip the downloaded file, and add the extracted bin folder as a path variable.

Alternatively, Gradle can be installed with Chocolatey:

```
$ choco install gradle
```

Linux

I've found that SDKMAN!, a utility to manage different versions of software development kits, is the easiest way to install Gradle. First, run this command to install SDKMAN!:

```
$ curl -s "https://get.sdkman.io" | bash
```

Then, run the following command to install Gradle:

```
$ sdk install gradle
```

Getting started with Rapture

Start by cloning the Rapture GitHub repo in a parent folder of your choosing (this parent folder will house all of your Rapira code):

```
$ git clone https://github.com/mattmikolay/rapture.git
```

> Note: you can also use the URL `https://github.com/rapira-book/rapture.git` (fork of original repo for archival purposes) if the URL above doesn't work.

Next, navigate into the newly created rapture directory and install rapture!

```
$ cd rapture && gradle installDist
```

"Hello, World!"

Let's write our first Rapira program! First, in the parent/root directory, we'll have to create a .rap file, which is the extension used for Rapira source code files. This file will be named `helloworld.rap`.

Open the folder and `helloworld.rap` in the text editor of your choice (I recommend VSCode). Then, write the following code:

```
output: "Hello, world!"
```

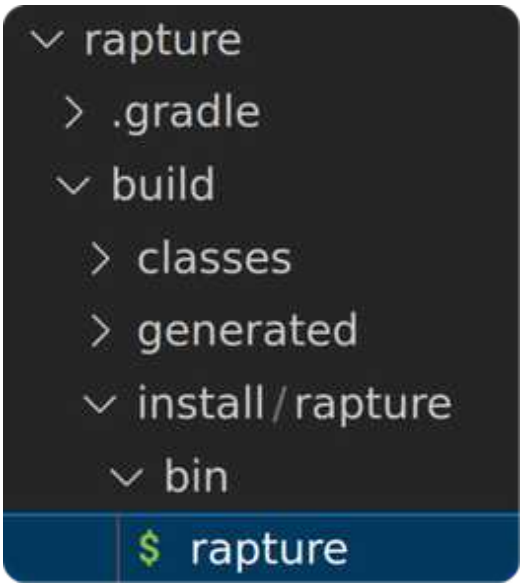
That's it, that's all it takes to print "Hello, world!" to the console! To run this code with Rapture, execute the following command:

```
$ rapture/build/install/rapture/bin/rapture helloworld.rap
```

Notice that we give the path of the Rapture executable file that was created when we ran the Gradle command, after which we give the name of the file we want to execute (in this case, `helloworld.rap`).

An optional (but helpful) tip

Listing out the entire Rapture path every time we want to run a Rapira program can be a bit tedious. Instead, we can create an alias for this command. To do this, we first need to get the absolute path of the Rapira executable on our machine.



On VSCode (and in several other text editors), you can right-click the highlighted executable file above, and select the "Copy path" or "Copy absolute path" from the dropdown menu. For example, on my Ubuntu machine, the absolute path is: `/home/amanvir/Code/rapira-code/rapture/build/install/rapture/bin/rapture`.

To be able use a command like `rapture helloworld.rap` instaed of stating the entire path, you need to open the config file for your shell (.bashrc, .zshrc, etc.).

Run the following command to open your config file (I'll be demonstrating with Bash):

```
$ nano ~/.bashrc
```

I've used nano, a lightweight CLI text editor, to open the bash config file, but you can choose to use vim, emacs, or even VSCode if you'd prefer. Once the file is open, add this line towards the bottom of your file, pasting the absolute path you copied earlier after the '=':

```
# Rapture alias
alias rapture=/home/amanvir/Code/rapira-code/rapture/build/install/rapture/bin/rapture
```

Save and exit from the file, and refresh the shell to see the changes take effect:

```
$ exec
```

Tada! You can now execute a much cleaner and shorter command (`rapture helloworld.rap`) in order to run some Rapira code!

Variables

Storing any kind of information is one of the most fundamental topics in computer programming; what would we do if we couldn't store numbers, letters, and more?!

That's where variables come in.

In Rapira, variables are declared like so:

```
num := 5
```

In this line, I just declared a variable called `num` that is being assigned the value `5`. Keep in mind that, in Rapira, the assignment operator is `:=`.

Also, notice how there is no keyword for the type of the variable (like `int`, in this case). Rapira uses dynamic typing like Python and Javascript in contrast to languages like C/C++ and Java, which utilize static typing (i.e., where variable types are known at compile time).

But, that's all there really is to instantiating a variable in Rapira! Here's some other types variables you can declare in Rapira:

```
\ Declares string named str, assigns value "hello" to
it
str := "hello"
```

```
\ Declares real named decimal, assigns value 0.5 to it
dec := 0.5
```

```
\ Declares boolean named bool, assigns value yes to it
\ Note: booleans can either be yes or no in Rapira
bool := yes
```

> Note that any statement followed by a `\` is a Rapira comment; I'll be using comments throughout the book to provide descriptions of code.

You've likely noticed by now that we use this syntax `output:` to print any data to the console. In similar fashion, we can actually use `input:` to take any input from the user and do stuff with it!

Here's an example where we use both output and input statements to get a number from the user, and print out the number's opposite to the console:

```
output: "Please enter a number"
input: n
output: "The opposite of your number is ", -n
```

This is what we get!

```
$ rapture input.rap
```

```
Please enter a number
: 2
The opposite of your number is -2
```

Nice! It works great! Here's the gist of whats going on: - Essentially, the variable n is being assigned the value of whatever the user inputs. We use this type of syntax instead of the `:=` operation when we specifically want user input. - Keep in mind that, since there are no types or type checking in Rapira, a user could enter any type of data (even a string, per se) and get away with it. This would lead to errors in our code when `-a` is executed, since you can't get the "opposite" of a string! We don't really have to worry about checking for types as of right now, but keep in mind that the code we've just written is susceptible to errors if we don't check that the right data is being passed in. - Notice how commas are used to separate multiple types of data when printing to the console (namely strings and integers in this case).

We've seen the "opposite of" operation on integers in this, but to end off, here's how you can execute common operations associated with integers in Rapira!

Integer Operation	Rapira Syntax
Addition	`a + b`
Subtraction	`a - b`
Multiplication	`a * b`

Integer Operation	Rapira Syntax
Division	`a / b`
Opposite	`-a`
Exponentiation	`a ** b`
Comparison	`a > b` or `a < b`
Equal to	`a = b`

In the next chapter, we'll be exploring operations with tuples and their operations!

Tuples

Need to store multiple elements of any kind in one variable? Use tuples!

To declare an empty Rapira tuple (called `t` in this case), we'll use this syntax:

```
t := <**>
```

The beginning of a tuple is marked by `<*` and ends with `*>`. To include values in this data structure, simply separate them with commas as such:

```
t := <* 1, 2, 3 *>
output: t
```

Printing out the contents of this tuple would yield `<1, 2, 3>` to the console.

To the already experienced programmers, tuples might seem eerily familiar to arrays, which are sets of contiguous elements that are all of the same type. In contrast to arrays, however, the same tuple can contain elements of varying types. For instance, this is a valid Rapira tuple that contains an integer, string, real, and boolean!

```
t := <* 1, "a", 0.5, yes *>
```

Indexing into a Tuple

Often, we'll want to "index into" the tuple, or, in simpler terms, access a specific element at a given index of a tuple. To do so in Rapira, we use the following:

```
t := <* 1, "a", 0.5, yes *>
output: t[2]
```

`2` is the index we are trying to access. Indices of a tuple start from 1 and continue on till the length of the tuple. So, all valid indices of a tuple range from `1` to `the tuple's length`. Therefore, trying to access the element at index `2` would print:

```
a
```

Pretty neat!

Getting the Length of a Tuple

If we populate a tuple, we'll often find that, at some point, we need to check how many elements are in our tuple.

In Rapira, this is extremely simple: just add a `#` before the name of the tuple. If we use the tuple we created as an example earlier,

```
t := <* 1, "a", 0.5, yes *>
```

the following code

```
output: #t
```

would print out `4`!

Prefixing and Postfixing

What if we wanted to get all the elements in an array after a certain index? Or, perhaps, before a certain index?

This could be extremely useful if we were using a tuple to model a queue of people. How can we get the names of all the people from position 3 till the end of the queue?

Simple! We use the postfix operator with a tuple in Rapira. Let's say our `queue` looked like this (starting with John and ending with Mary.

```
queue := <* "John", "Amy", "Max", "Oscar", "Mary" *>
```

We can specify that we want to print out the names of all the people at position 3 onwards by using the same notation for indexing into a tuple, but this time, adding the : character directly after our specified index (3, in this case):

```
queue[3:]
```

When printed, this statement will output the following to the console:

```
<* "Max", "Oscar", "Mary" *>
```

Looking back at our tuple, we can confirm that "Max", "Oscar", and "Mary" are the names of the people from position 3 till the end of th equeue. The code works! Postfixing is so cool!

Now what about the second scenario, where we'd want to get all of the names of people before a certain index. In the case of our queue, what if we wanted to get the names of the first 3 people in the queue.

Easy! We use the prefix operator, which works identically to postfix, except the : goes before the index. So, to get the names of the first 3 people waiting in the queue, we'd use the following code:

```
queue[:3]
```

This would print:

```
<* "John", "Amy", "Max" *>
```

Nice!

But here's the kicker. In both of the cases we've just explored, we saw how to print all elements from a certain index and either the beginning or the end of the tuple. But what if we wanted to print out the names of all the people in the queue excluding the first and last persons in the queue. In other words, we'd want to print the names of all people from index `2` to the index at the `length of the queue - 1`. With our queue, the code would look like this:

```
queue := <* "John", "Amy", "Max", "Oscar", "Mary" *>
output: queue[2:(#queue - 1)]
```

Yay! We've done it. The following code outputs:

```
<* "Amy", "Max", "Oscar" *>
```

Look to the next chapter to learn about strings in Rapira! Tuples and strings work in a similar fashion, and because you already know about tuples, using strings will be a cakewalk! Let's go!

Strings

Often, we'll need a way to store and manipulate text in Rapira. So, we'll need to know how strings, contiguous sequences of characters, work in Rapira. Let's start with how to initialize an example string.

```
str := "Hello World!"
```

As you can see above, a set of double quotes are used to mark the beginning and end of a string. We'll now look at methods that Rapira offers for strings, several of which are very similar to the methods for tuples!

Indexing into a String

```
str := "abcdef"
output: str[2]
```

Seem familiar? It's practically the same syntax for indexing into a tuple. The code above would output `a` to the console.

Getting the Length of a String

It's the same as tuples!

```
str := "abcdef"
output: #str
```

Running this, you should see `6` printed to the console!

Prefixing and Postfixing

Prefixing looks like this (prints `fly`):

```
str := "butterfly"
output: str[7:]
```

Postfixing is accomplished like so (prints `butter`):

```
str := "butterfly"
output: str[:6]
```

Again, it's pretty much the same as prefixing and postfixing with tuples! But hold on tight, we're going to see a new method now: `index()`!

Getting the Index of a Character

We've established how you can get the character at a particular index, but how can you get the index of a character in a string?

Use the `index()` method! Let's check out the following example:

```
str := "butterfly"
idx := index("f", str)
```

The resulting value of `idx` is `7`, as that's the index of the *first*, and, in this case, the only occurrence of the letter `f`.

If we wrote the following line instead:

```
str := "butterfly"
idx := index("t", str)
```

What do you think the console would read after printing idx? I'd urge you to try it out on your own!

If you guessed that `3` would be outputted to the console, you'd be correct! Remember, it's the first instance of the corresponding character whose index is returned by the function.

`If/else` statements

So far, we've been executing statements in Rapira without changing the control flow of the program. We particularly haven't been using `if/else` statetements or loops (we'll cover the latter in the next chapter). But what are `if/else` statements? Well, let's look at `if` statements to begin with.

`If` statements

`if` statements contain a boolean condition; if the boolean condition evaluates to `yes`, then the code wrapped inside the `if` statement is executed. The code wrapped inside the `if` statement is **not** executed if the boolean condition evaluates to `no`.

Let's take a look at an example to understand this more:

```
num := 5
if num > 3 then
  output: "num is greater than 3!"
fi
```

In the first line, we instantiate a variable num and assign it the value `5`. In the following lines, the real magic happens.

Using the keyword `if`, we start a Rapira `if` statement! Then, nestled within the keyword `if` and `then`, comes the boolean

condition (which, as stated earlier, evaluates to either `yes` or `no`): in our case that's `num > 3`. So, if `num > 3` evaluates to `yes`, we enter the body of the `if` statement. In this case, you'll notice that the condition *does* evaluate to `yes`... we continue to the code in the body then!

The statement `output: "num is greater than 3!"` executes, and the given code is printed to the console! It should be noted that the code in the body of the `if` statement is indented. The next line contains the `fi`, the reverse of `if`, thereby signifying the end of the `if` statement!

Congratulations! You've just learned what an `if` statement is and how it works!

`Else` statements

While we're here, let's quickly look at `else` statements with this code snippet:

```
str := "foo"
if #str > 3 then
  output: "str is more than 3 characters in length!"
else
  output: "str is less than or equal to 3 characters
in length!"
fi
```

By using the keyword `else` and adding any indented code after it, we've effectively added code that evaluates whenever the

boolean condition turns out to evaluate to `no`. Like in this case, where the `else` code is executed and `str is less than or equal to 3 characters in length!` is printed to the console!

Loops

While `if/else` statements change the control flow by only conditionally executing some code, loops change the control flow by repeating a block of code a certain number of times.

In Rapira, there's two types of loops. Let's start by looking at `for` loops.

`for` loops

`for` loops can be used to execute a block of code a predefined, set number of times. In Rapira, they take in three values: the name of a counter variable that is incremented, a minimum value, and a maximum value.

Let's see this in action with a tuple `t` with all integer elements. We'll try to use a `for` loop to find the sum of all elements in the tuple:

```
t := <* 1, 2, 3 *>
sum := 0

for N from 1 to #t do
  sum := sum + t[N]
od

output: sum
```

As you can see, on line 3, we begin to declare the `for` loop with the `for` keyword. We then list a variable we'd like to use to keep track of how many iterations we have already executed by incrementing it by `1` with every iteration; in this case, that variable is `N`. Following that, we declare the minimum value that `N` is initially set to (`1`) and the maximum value that, when `N` is equal to it, determines the stopping point for the looping (`#t`, which is 3). Thus, `N` is initially set to `1`, but is then incremented to `1` and then finally incremented to `3` in the 3rd iteration.

In the body of the loop above, you can see that with t[`N`], we can get the value of each element in the tuple, iteration-by-iteration. This allows us to effectively add each value to a sum variable and eventually print it after the loop finishes:

```
6
```

`while` loops

Although `for` loops rely on a set number of predetermined iterations, `while` loops continue to execute a block of code repeatedly so long as the provided condition is true. Think of it as an `if` statement, except that the code is repeated until the condition turns out to be false, which, keep in mind, could be never!

Here's an example of the previous example rewritten to use a `while` loop:

```
t := <* 1, 2, 3 *>
sum := 0
counter := #t

while counter > 0 do
  sum := sum + t[counter]
  counter := counter - 1
od

output: sum
```

Functions

You're almost done with your journey of learning Rapira! Let's finish up with functions.

In general, functions are grouped blocks ("chunks") of code that perform a particular task. Therefore, they can be repeatedly called, causing the code they encompass to be executed repeatedly.

Below is function called `printFive()` which does exactly what you think it does: it outputs `5` to the console!

```
fun printFive()
  output: 5
end

printFive()
printFive()
printFive()
```

Notice that a function is defined with the `fun` keyword, following which you list out its chosen name and a set of parentheses. Between this line and the `end` keyword is the function's code. In this case, it's simply `output: 5`.

The next three lines proceed to call the function, which leads the following to be printed to the console:

```
5
5
5
```

That's great!

But this is only a simplistic view of functions. Let's look at a more complicated case: say you were tasked with finding the arithmetic mean of all elements in a numerical tuple using a function.

We'll have to expand on our knowledge of functions in order to tackle this problem. We need to pass in an argument into the function. Namely, it's the tuple we'd like to use!

Here's how you write a function with an argument:

```
fun mean(t)
  sum := 0
  for N from 1 to #t do
    sum := sum + t[N]
  od
  return (sum / #t)
end

output: mean(<* 1, 2, 3, 4, 5 *>)
```

The function declaration is not that much different than the one from our last example. All we've done is added a "variable" of sorts (`t`) between the two parentheses at the end of line 1. This is our argument. It functions identically to a

variable,except that its scoep is restricted to the function its defined in.

It's value is determined at the final line of the code. When we write

`mean(<1, 2, 3, 4, 5>)`, the tuple `<1, 2, 3, 4, 5>` is passed in as input to the function. We use `t` within the scope of the function to access this tuple!

Challenges

Solve these challenges in Rapira! Refer to the sample answers in the next chapter once you've solved the problem or if you're stuck.

1. Even or odd

Test whether an integer is even or odd. Hint: use the modulo operator (`/%`).

2. Factorial

Write a function to return the factorial of a number.

3. Fibonacci

Write a function to generate the nth Fibonacci number.

4. Input loop

Read from a text stream line-by-line. The stream will have an unknown amount of data on it.

5. Leap year

Determine whether a given year is a leap year in the Gregorian calendar. There is a leap year when the year is perfectly divisible by 4, except when the year is divisible by 100 and not divisible by 400.

6. Nested loops

Using a loop (or multiple loops), print out the following pattern:

```
*
**
***
****
*****
```

Sample Solutions for Challenges

Cross-check with these example solutions!

1. Even or odd

```
fun is_even(n)
  return (n /% 2) = 0
end
```

2. Factorial

```
fun factorial(number)
  if number = 0 then
    return 1
  fi

  return number * factorial(number - 1)
end
```

3. Fibonacci

```
fun fibonacci(n)
  if n = 0 then
    return 0
  fi
  if n = 1 then
    return 1
  fi
  return fibonacci(n - 1) + fibonacci(n - 2)
end
```

4. Input loop

```
while 1 do
  input text: str
  output: str
od
```

5. Leap year

```
fun is_leap_year(year)
  if (year /% 100) = 0 then
    return (year /% 400) = 0
  fi
  return (year /% 4) = 0
end
```

6. Nested loops

```
for N from 1 to 5 do
  output: "*" * N
od
```